

**Managing Embedded System Software**  
with the  
**RFIES Development Tools**

## Table of Contents

1.0 Background - The Wind River Tools.....	3
2.0 The RFIES Software Development Model .....	3
3.0 Projects.....	5
3.1 Front-end Projects .....	6
3.2 Shard Library Projects .....	6
4.0 The Embedded System Development Tools .....	7
4.1 Tailoring the Tool Environment .....	7
4.2 Setup Tool – use: setup projectName [targetName [headerType] ] .....	9
4.3 Target Tool – use: target [targetName].....	11
4.4 Headers Tool - use: headers [production test development] .....	12
4.5 Help Tool - use: helpme.....	12
5.0 Building Software Modules with make .....	14
5.1 Make Parameters.....	16
5.2 Make Rules .....	17
6.0 Environment Variables .....	19
6.1 Developer Supplied Environment Variables .....	19
6.2 Utility Environment Variables .....	19

## **1.0 Background - The Wind River Tools**

The Accelerator Division uses Wind River's Tornado, an integrated software development environment for VxWorks, to develop software for its control system front-end computers. Tornado resides on the development node nova.fnal.gov and employs the GNU compiler, linker and support utilities to translate C and C++ source code files into object modules suitable for execution on the targeted front-ends. Tornado's GNU tools are referred to as cross development tools because they are designed to generate code on one machine, in this case nova, that is intended to execute on another machine of different architecture, the front-end. Various shell environment variables and tool command line arguments configure the GNU tools for the code translation process. Such attributes as target CPU instruction set, code optimization level and object file format must be specified.

The AD Controls Department provides a family of shell scripts to configure the GNU tool environment variables for the various front-end architectures used within the control system. These scripts are easily accessible through 'env\_xxx' shell aliases which are exported to the developer's environment when logging-in to nova. Many of the necessary GNU tool command line arguments are uniquely tied to front-end code performance and must be provided by the developer. GNU's powerful make utility is employed to orchestrate the code translation process by invoking the various other tools in the correct sequence and with the appropriate command line arguments. To accomplish this the make tool processes a collection of rules, dependencies and value definitions provided by the developer in a text file called Makefile.

To generate executable code for any given target system the developer must write a Makefile appropriate for the target's architecture, invoke the proper env\_xxx shell alias to configure the GNU tools and finally invoke GNU make to run the compiler and linker against project source code files. Furthermore, if the project software supports multiple target architectures, as is the case for many general purpose shared libraries, this process must be repeated for each one of the targets. Most of this tedious process is automated by the RFIES embedded software development tools.

## **2.0 The RFIES Software Development Model**

The RFIES embedded software development tools (hereafter referred to as "the tools") support a set of project management practices that ensure a stable and consistent mechanical process for developing front-end software. Software is developed under a three release level model. The three release levels are referred to as: '**development**',

**‘test’** and **‘production’**. The tools assist the developer by automating the mechanics of the process at each level.

Development level software is generally in the early stages of development, or may be mature code that is being updated with new features or improved in some other way. At this level the code is usually in a repeated edit – compile - test phase and is not ready to be presented to users. Testing at this level is at the unit level and often involves integration testing with other related software modules. When the software is felt to be feature complete and has passed all integration testing it is ready to be promoted to the test level. The development level phase does not require that the code be committed to the repository.

Test level software has undergone rigorous unit level and integration testing and is ready to be tested in the target environment. Often this is referred to as beam or machine testing. This is the first time the code is run with related production level modules under actual operating conditions. Many times the complexity of the system is high and testing under simulated conditions is not practical or, for that matter, possible. If test level operation is unsuccessful the code is returned to the development level for further work. The test level phase does not require that the code be committed to the repository.

When the code has been demonstrated to be correct and is accepted by project management, it is ready to be promoted to the production level. The project code must be committed to the repository before it can be promoted to the production level. After committing the project to the repository a make production command will install the final product in the appropriate production destination directory. See the note “[Creating CVS Projects](#)” for details on placing project code into the CVS repository.

Software that is at any of the three levels of release may be compiled and tested against other modules that are themselves at any of the three release levels. Some combinations may be meaningful, others not so much. It is up to the developer to understand the dependencies and interactions between various modules at various release levels. Progression from one release level to the next is supported by the tools but not enforced. It is up to the developer to be fastidious about following the three level development/release cycle.

### 3.0 Projects

A **‘project’** is a functionally cohesive collection of one or more software modules containing source code files any related support material. There are shared library projects that contain low level support features such as I/O drivers, and there are front-end projects containing top level software that may use zero or more of the shared libraries for support. There is really little technical difference between the two project types other than the way that they are linked and deployed. A third project type called a dsp project is nearly identical to the shared library project differing mainly in the way the library is installed.

By convention projects are given a name (in all lower case) relating to their intended functionality. For example the miscvxworks project contains miscellaneous support code for the VxWorks kernel while the millrf project contains the Main Injector Low Level RF front-end code. A list of all existing project names can be obtained with the **‘listp’** tool, and you can determine if some derivative form of a name string has already been used in the software repository with the **‘incvs’** tool.

By convention projects are developed in a sandbox directory having the same name as the project and located within ~/esd/src. At developer determined milestones the contents of this sandbox are committed to the code repository for version control and safe storage. New project templates are created automatically with the **‘setup’** tool. Setup is also used to check existing projects out of the software repository and for rapid movement between projects within the developer’s sandbox area. See section 4.2 for more details on the setup tool.

Every project must contain a **‘Targets’** file which lists all possible build targets, and a **‘Makefile’** file that specifies special project build dependencies and any target specific specialization. Both files are treated as any other source code file and reside with the project’s source code files in the sandbox directory. See section 4.3 for a description of the Targets file and section 5.0 for a description of the Makefile file.

When a project is released at a given level its public header files are placed in a common header file pool and, if a shared library, its object module is stored in a common library file pool. Front-end object modules are always stored in the front-end’s unique download directory. Shared library header and object pool directories have the release level encoded in their directory path. Front-end object modules have the release level encoded in their file name proper. See sections 3.1 and 3.2 for more information on object module installation.

Projects are built against other projects at any one of the three release levels and then installed at any one of the release levels. For example one could build a development level library against test level support libraries. The **‘dev[elopment] ‘, ‘test‘**, and **‘prod[uction] ‘** commands configure the release level of reference files and support libraries while the **‘make development‘**, **‘make test‘** and **‘make production‘** commands build the project at the associated level. See section 4.4 for a description of the development, test and production command aliases, and section 5.2 for a more on the make install process.

### 3.1 Front-end Projects

A project is configured as a front-end project by including the line:

```
#include $(ESD_INC_DIR)/download.mk
```

in the project’s Makefile. Front-end projects are special in that their object modules are installed (on nova) in the /fencode-bd/vxworks\_boot/fe/xxx directory tree. By convention the front-end’s network node name, ACNet node name and project name are all identical containing six or less lower case characters. For any project xxx the make development, test and production commands place devxxx.out, testxxx.out and libxxx.out, respectively, in the /fencode-bd/vxworks\_boot/fe/xxx directory. This directory should contain any front-end related script and configuration files. A sister /fencode-bd/vxworks\_write/fe/xxx directory supports front-end created files if necessary. Any public header files will be installed in the same directories used by shared libraries as described in section 3.2 below. See the note [“Downloading Front-ends from fencode-bd”](#) for details on creating and using front–end download directories.

### 3.2 Shared Library Projects

A project is configured as a shared library project by including the line:

```
#include $(ESD_INC_DIR)/install.mk
```

in the project’s Makefile. Shared library projects are perhaps the most common. Shared library object modules are installed (on nova) in the /fencode-bd/vxworks\_boot/fe/ORG/LIB directory tree where ORG represents the developer’s organization (e.g., rfies, rfiinst) and LIB is one of devlib, testlib or lib as appropriate. Public header files for the library are installed (on nova) in the /home/rfies/esd/ORG/INC directory tree where INC is one of devinc, testinc or inc as appropriate.

## 4.0 The Embedded System Development Tools

The tools assume the bash shell – use of other shells is not recommended. Developers can temporarily switch to the bash shell by simply calling it from any other shell. If bash is not your default shell, making it so will require the assistance of nova's System Administrator.

Developers who are new to nova can easily configure new accounts for use of the tools by running the **'useresdconfig'** script to automatically create the standard developer account directory structure and obtain copies of all required configuration files. A related note "[Configuring Accounts on Nova](#)" describes the procedure for configuring your account for use of the tools.

The tools are located in the /home/rfies directory on nova. The software repository used by the tools is managed by CVS with its exact location transparent to developers. The default configuration of the tools supports the UN\*X bdrfinst group, but members of other UN\*X groups may tailor the tools as described in section 4.1 below.

### 4.1 Tailoring the Tool Environment

Developer access to the tools is established by copying the file **'/home/rfies/esd/examples/useresdsetup.bash'** to a local scripts directory, optionally editing the copy to establish any unique requirements, and executing the updated copy with the shell's source command. There are four definitions in the useresdsetup.bash file that may be modified by the developer:

- **USER\_ORGANIZATION** must contain the developer's organization name. This variable helps specify the project location within the source code repository and installation directories. Options include rfies and rfinst – others may be added in the future. By default USER\_ORGANIZATION is rfies.

- **USER\_SCRIPT\_DIR** must point to a directory containing 'callback scripts' used by the tools to tailor tool operation to meet special project needs. By default USER\_SCRIPT\_DIR is set to ~/esd/scripts.

- **USER\_SANDBOX\_DIR** must point to a directory containing one subdirectory for each of the developer's projects. The convention is to have a subdirectory for each project with the same name as the project (i.e., project foo would be in a directory called \$USER\_SANDBOX\_DIR/foo.) By default USER\_SANDBOX\_DIR is set to ~/esd/src.

- **USER\_REPOSITORY** must contain either 'CVS' or 'SCCS'. This variable tells the tools which source code repository the developer prefers and allows the tools to create aliases to simplify use of the repository. By default USER\_REPOSITORY is set to 'CVS'.

Figure 1 represents an example `usersdsetup.bash` script containing definitions for a developer who happens to be providing values identical to the defaults. By the way, the name `usersdsetup.bash` is not sacred, developers may rename their copy as desired because they are the only ones referencing this file.

```
#!/usr/local/bin/bash
#
# Id: %M% %I% %G% %U%
# Id: $Id$
#
# Description:
#   Script to set up the rfies group's embedded system
#   development tools on behalf of the developer.
#   This script should be sourced by the caller.
#

#
# set up the developers' development organization
#
export USER_ORGANIZATION=rfies

#
# set up where tools find:
#   project_xxx scripts
#
export USER_SCRIPT_DIR=~/esd/scripts

#
# set up where tools find:
#   directories, with same name as projects, which contain project source
#
export USER_SANDBOX_DIR=~/esd/src

#
# set up developers repository preference (e.g., CVS, SCCS or SCM)
#
export USER_REPOSITORY=CVS

#
# set up embedded system development tools
#
source /home/rfies/esd/scripts/esdsetup.bash

#
# End of script
#
```

**Figure 1**

The final operation in the `usersdsetup.bash` file is to run the global tools configuration script, once the script has been run the tools and software repository will be fully configured and available for use.

#### **4.2 Setup Tool – use: `setup projectName [targetName [headerType] ]`**

The '**setup**' tool activates a project for development. In its most basic form: **setup projectName**, setup is used to navigate to a project sandbox directory and configure the tools for that project. Setup must be used to initially move to a project or to move from one project to another. The optional **targetName** parameter specifies the target to be built in subsequent make operations. If **targetName** is not specified the most recently built target will be used. If the project sandbox is clean (i.e., no previous build, or the project has been made clean) the first target listed in the Targets file will be used. The optional **headerType** parameter specifies the release level of referenced header and object file pools used in subsequent make operations. If **headerType** is not specified the most recently referenced files will be used. If the sandbox is clean the production reference files will be used. NOTE: If use of the **headerType** parameter is desired then the **targetName** parameter must also be specified in the order shown above. See sections 4.3 and 4.4 for an alternative and more commonly used mechanism for specifying target and header types.

If the setup tool does not find **projectName** in the developer's sandbox it will look in the software repository and if found there, setup will offer to check the project out into the correct sandbox. If **projectName** is not found in the repository setup will offer to create a new project with that name in the developer's sandbox directory. This scheme allows new projects to be created with the setup tool by simply typing '`setup newProjectName.`' In this situation setup will ask for the project type and offer to install template project files.

In summary the setup command does the following for the developer:

- 1 - configures environment variable **PROJECT**,
- 2 - configures environment variable **USER\_PROJECT\_DIR**,
- 3 - configures environment variable **TARGET**,
- 4 - sources `$USER_SCRIPT_DIR/project_projectName` iff it exists,
- 5 - creates handy aliases for using the repository with the project,
- 6 - does a `cd` to the project sandbox,
- 7 - runs the target tool and
- 8 - runs the headers tool.

For item #4 above the script \$USER\_SCRIPT\_DIR/project\_projectName is optional and is used to specify overriding values for the \$USER\_PROJECT\_DIR, \$TARGET and/or \$HEADERS environment variables, and to define project specific environment variables. Setting \_USER\_PROJECT\_DIR\_ allows the project's source code files to reside in a location other than the customary location: \$USER\_SANDBOX\_DIR/\$PROJECT. Setting \_TARGET\_ will specify a target for all builds. Setting \_HEADERS\_ will specify which reference files to compile against for all builds. An example project\_projectName file may be copied from **\$(ESD\_BASE\_DIR)/examples/project\_xxx**. Figure 2 represents an example project\_projectName script for a project with the source code files in the standard location and to be built for the PPC603 target and against production referenced headers and object files with the shell variable \$PROJECT\_FLAGS initialized to 123.

```
#!/usr/local/bin/bash
#
# Id: %M% %I% %G% %U%
# Id: $Id$
#
# Description:
#   Set up environment for project xxx.
#   Sourced by the ESD_SCRIPT_DIR\setup.bash script.
#
#
# developers may specify the source directory for $PROJECT
#
_USER_PROJECT_DIR_=$USER_SANDBOX_DIR/$PROJECT
#
# developers may specify the target which they want to build by default
#
_TARGET_="PPC603"
#
# developers may specify the headers that they want to use by default
#
_HEADERS_=production
#
# developers may specify any project specific shell variables
#
PROJECT_FLAGS=123
#
# End of script
#
```

**Figure 2**

### **4.3 Target Tool – use: target [targetName]**

The **'target'** tool allows developers to specify a new target for subsequent makes. The tool searches the Targets file (described below) for a specification matching targetName and if a match is found initializes the tool environment for that target. The target tool also places the specified target name in a file named Target to act as a configuration variable with persistence. Since the Target file is in the make dependency list for all object modules this will guarantee that the project will be rebuilt whenever the target changes.

The Targets file has the following format:

```
# Targets for project
targetName[-targetVariant][_buildVariant] setupScriptName [flag...]
```

Each line of the Targets file not beginning with a # represents a target definition containing two or more fields. Figure 3 represents a typical Targets file content.

```
#
# Id: %M% %I% %G% %U%
# Id: $Id$
#
# format TARGET[-targetVariant][_buildVariant] (in that order)
# e.g., MVME5500-rrbpm_61
# The first target listed will become the 'default'
#
MVME5500_64 /usr/local/vxworks/scripts/wind6.4_mv5500.bash
MVME2434_64 /usr/local/vxworks/scripts/wind6.4_mv2434.bash
MVME5500_61 /usr/local/vxworks/scripts/wind6.1_mv5500.bash
MVME2434_61 /usr/local/vxworks/scripts/wind6.1_mv2434.bash
MVME5500_55 /usr/local/vxworks/scripts/wind55_mv5500.bash
MVME2434_55 /usr/local/vxworks/scripts/wind55_mv2434.bash
```

**Figure 3**

The targetName field is required and gives a name to the target being defined. Target names are all capitalized by convention and generally reflect some significant characteristic of the target such as its CPU or SBC (e.g., PPC603 or MVME5500.) The optional –targetVariant extension to targetName sets a make variable to ‘VARIANT=targetVariant’ and a compile-time definition ‘-DVARIANT targetVariant’ in support of conditional compilation and target variant installs. For front-end projects, object modules will be installed into /fencode-bd/vxworks\_boot/fe/targetVariant, and for shared library

projects the library name will have `-targetVariant` appended (e.g., `libmiscvxworks-targetVariant.out`.) The optional `_buildVariant` extension to `targetName` provides for building the target with different tool configurations by defining multiple target definition lines each with a different `_buildVariant` / `setupScript` pair.

The `setupScriptName` field is required and contains the file name specification for the setup script that configures the tools for the specific processor and version of VxWorks (e.g., `/usr/local/vxworks/scripts/wind6.1_mv5500.bash`.) The various forms of this file name specification can be determined by typing the following shell command:

```
alias | grep env
```

The `flag` field represents zero or more flags of the form `xxx=yyy` that will be passed to `make` on the command line whenever the target is built.

#### 4.4 Headers Tool - use: `headers [production|test|development]`

The **'headers'** tool places the specified reference file release level in a file named `Headers` to act as a configuration variable with persistence. Each time a target is built the tools use the `Headers` file to determine which production level reference files to use. If the `Headers` file is not present then the setup tool or make tool will initialize it to `'production'`. Since the `Headers` file is in the dependency list for all object modules this will guarantee that the project will be rebuilt whenever the headers value changes. For convenience the following shorthand aliases are provided:

<u>Alias</u>	<u>Equivalent headers command</u>
<code>'production'   'prod'</code>	<code>'headers production'</code>
<code>'test'</code>	<code>'headers test'</code>
<code>'development'   'dev'</code>	<code>'headers development'</code>

#### 4.5 Help Tool - use: `helpme`

Since additional tools are likely to be added to the system over time the **'helpme'** tool prints terse but hopefully helpful hints about the various tool commands and associated parameters. Figure 4 contains typical text produced by the `helpme` tool.

```
builddevelopment - do a make development for each target specified in Targets
buildnull - do a make without install for each target specified in Targets
buildproduction - do a make production for each target specified in Targets
buildtest - do a make test for each target specified in Targets
dev|development - switch header file references to development for future makes
dumpesd - list all development tool related environment variables
```

finddsp fileName - find specified file in DSP directories (dsp, testdsp and devdsp)  
 findinc fileName - find specified file in include directories (inc, testinc and devinc)  
 findlib fileName - find specified file in library directories (lib, testlib and devlib)  
 findproducts fileName - find specified file in PRODUCTS\_INCDIR  
 findsrc fileName - find specified file in user's sandbox directory  
 findvx fileName - find specified file in VxWorks directory structure  
 headers [production|test|development] :  
     headers - display header library referenced for subsequent compiles  
     headers [production|test|development] - set up header library references for subsequent compiles  
 helpme - list help (this list) for ESD tools  
 incvs pattern - check to see if 'pattern' is part of any project name in the CVS repository  
 listp [orgName] :  
     listp - list all projects in the repository belonging to group rfiinst  
     listp orgName - list all projects in the repository belonging to members of the specified organization  
 lists - list all projects in the user's sandbox  
 listt - list all targets for the current project  
 make [project=xxx] [target=yyy] [headers=production|test|development] [file=dspObjectName] [makeTarget] :  
     make help - print help for makeTarget rules  
     make install - print help for make install rules for current target  
 prod|production - switch header file references to production for future makes  
 readme - do a more of any README file in the cwd  
 search searchPattern [dsp|fe|shared|all] - search source repositories for definition of searchPattern  
 searchheaders searchPattern [inc|testinc|devinc] - search rfies headers for (grep style) searchPattern  
 searchproducts searchPattern - search PRODUCTS\_INCDIR headers for (grep style) searchPattern  
 searchsrc searchPattern - search user's sandbox directory for (grep style) searchPattern  
 searchvx searchPattern - search VxWorks headers for (grep style) searchPattern  
 setup projectName [targetname] [production|test|development] :  
     setup projectName - cd to projectName sandbox & set up default target  
     setup projectName targetName - cd to projectName sandbox & set up target targetName  
     setup projectName targetName production - cd to projectName sandbox & set up target targetName using production headers  
 target [targetName] :  
     target - display target for subsequent compiles  
     target targetName - set up target targetName for subsequent compiles  
 targs | targets - list all targets defined in the Targets file  
 test - switch header file references to test for future makes  
 xpv& - X Window Project Manager/Viewer (run in background)

**Figure 4**

## 5.0 Building Software Modules with make

The conversion of collections of C and C++ source code files into useful executable modules is accomplished with the aid of the GNU make utility. Make provides a mechanism for codifying the rules by which software modules are built in a Makefile, and then automatically applying those rules to invoke the language compilers and linkers as necessary to produce final object modules. Many software modules contain code that can be used in more than one application or be executed on more than one processor. With the help of make these general software modules can easily be rebuilt for each such target.

Tornado provides a set of make include files to simplify the process of creating Makefiles but unfortunately these files support only single target builds. The functionality of the Tornado make include files has been extended by the tools' build.mk make include file, to provide rules for compiling mixed C and C++ source code files and for creating object modules for any specified target. The tools' build.mk file is included by reference in the project Makefile which needs only to provide high-level specifications of target module composition. This allows the Makefile to be quite uncomplicated in appearance.

Figure 5 contains an example Makefile for a project supporting more than one processor and target. The example appears to be quite lengthy because it includes examples of definitions for multiple processors and targets. Careful inspection reveals that most of the definitions in the example are null and can be removed.

```
# Id: %M% %I% %G% %U%
# Id: $Id$
#
# Description:
#   Makefile for projects.
#

# specify sources which must be compiled
C++SOURCES = $(wildcard *.cpp)
CSOURCES = $(wildcard *.c)
# specify all header files to be installed in the includes directory
HEADERS = $(wildcard [!_]*.h)

# specify all startup script files to be installed in front-end
# download directory
SCRIPTS = $(wildcard *startup)

# specify compiler parameters which affect all builds
LIBRARIES =
INCLUDES =
```

```

DEFINES =
CFLAGS =
C++FLAGS = $(CFLAGS)

# specify additional parameters which affect specific versions of VxWorks
# where xxx is eg, 55 or 61
VW_xxx_C++SOURCES =
VW_xxx_CSOURCES =
VW_xxx_HEADERS =
VW_xxx_SCRIPTS =
VW_xxx_LIBRARIES =
VW_xxx_INCLUDES =
VW_xxx_DEFINES =
VW_xxx_CFLAGS =
VW_xxx_C++FLAGS = $(CPU_xxx_CFLAGS)

# specify additional parameters which affect specific processors
# where xxx is eg, MC68040 or PPC604
CPU_xxx_C++SOURCES =
CPU_xxx_CSOURCES =
CPU_xxx_HEADERS =
CPU_xxx_SCRIPTS =
CPU_xxx_LIBRARIES =
CPU_xxx_INCLUDES =
CPU_xxx_DEFINES =
CPU_xxx_CFLAGS =
CPU_xxx_C++FLAGS = $(CPU_xxx_CFLAGS)

# specify additional parameters which affect specific targets
# where xxx is eg, MVME2434 or MVME5500
TARGET_xxx_C++SOURCES =
TARGET_xxx_CSOURCES =
TARGET_xxx_HEADERS =
TARGET_xxx_SCRIPTS =
TARGET_xxx_LIBRARIES =
TARGET_xxx_INCLUDES =
TARGET_xxx_DEFINES =
TARGET_xxx_CFLAGS =
TARGET_xxx_C++FLAGS = $(TARGET_xxx_CFLAGS)

# specify additional parameters which affect specific variants
# where xxx is eg, recycler or debug
VARIANT_xxx_C++SOURCES =
VARIANT_xxx_CSOURCES =
VARIANT_xxx_HEADERS =
VARIANT_xxx_SCRIPTS =
VARIANT_xxx_LIBRARIES =
VARIANT_xxx_INCLUDES =
VARIANT_xxx_DEFINES =
VARIANT_xxx_CFLAGS =
VARIANT_xxx_C++FLAGS = $(VARIANT_xxx_CFLAGS)

# use character '@' for quiet makes, leave blank to detail make process

```

```

OUT = @

#
# End of developer portion of makefile -- include xxx.mk include files
below
#
include build.mk      # rules for building projects
include install.mk    # rules for installing projects into libraries

#
# End of makefile
#

```

**Figure 5**

An example Makefile may be copied from **\$(ESD\_BASE\_DIR)/examples/project/Makefile**.

The make tool passes the following definitions on the compiler command line so that your source code can determine the environment for which it is being compiled:

- -DCPU=wwwwww,
- -DOS\_VERSION=xxxx,
- -DTARGET=yyyy and
- -DVARIANT=zzzz

where wwwwww could be MC68020 or PPC603, xxxx could be VW\_55 or VW\_64, yyyy could be MVME2434 or MVME5500 and zzzz could be rtbpm or mbpm for example. The full set of CPU and OS\_VERSION definitions is dependent upon the available Tornado tools. The TARGET and VARIANT definitions are provided by the developer in the Targets file.

## 5.1 Make Parameters

The make tool supports optional command line parameters for modifying the default make process:

```

make [project=xxx] [target=yyy] [headers=production|test|development] \
[file=dspObjectName] [makeTarget]

```

The project parameter tells the make system which project is being built. The value provided will override the PROJECT environment value. This switch is intended for non-interactive use in other tools.

The target parameter tells the make system which target is being built. The value provided will override the TARGET environment value. This switch is intended for non-interactive use in other tools.

The headers parameter tells the make system which reference files to include:

- **headers=production** – use the production reference files.
- **headers=test** – use the test reference files.
- **headers=development** – use the development reference files.

If these switches are not used the default action is to use the production reference files.

The file parameter tells make explicitly which object file to install in cases where there may be more than one object file per project:

- **file=fileSpec[.ext]** – install the specified object file.

If this parameter is not used the default action is to install projectName.ldr. If the extension to the fileSpec operand is not specified .ldr will be assumed. This switch applies only to DSP object module install operations.

The makeTarget parameter tells make which of the standard make rules to process. See section 5.2 for a description of the make rules.

## 5.2 Make Rules

The make tool provides a set of rules for building projects and another set for installing projects.

The build rules provided by build.mk direct the compiler and linker to produce object modules supporting the specified CPU, OS version and target. The build rules include:

- **make** – Build (i.e., make all) the project for the currently specified target.
- **make clean** – Remove all generated files (i.e., .o, .a, .doc, and munching files) from the cwd.
- **make doc** – Make document file for all source code files in the project.
- **make echo** – Print a list of the project's header, source code and script files.
- **make help** – Print help information about the make rules.
- **make info** – Print information about the current project configuration.
- **make librarydirectory** – Create production, test and development library directories on fecode-bd for the current target. If the directories already exist nothing will be altered. This is useful when building for a previously undefined target.

- **make lint** – Run the lint program on all project C and C++ source code files.
- **make map** – Produce linker map file.
- **make <file>.cppsym** – List all preprocessor symbols for the specified file.
- **make <file.ext>.doc** – Make document file from the specified file.
- **make <file>.lint** – Run the lint program on the specified file.
- **make <file>.out** – Compile and munch single file into <file>.out.
- **make <file>.pp** – Produce preprocessor output only for the specified file.
- **make <file>.s** – Produce assembly source listing for the specified file.

The install rules, provided by install.mk and dspinstall.mk, use shell commands to copy the project's header and object files into the appropriate libraries. The install rules include:

- **make downloaddirectory** – Create a download directory on fecode-bd for the current project. If the directory already exists nothing will be altered. This is useful when setting up a new front-end for downloading.
- **make librarydirectory** – Create production, test and development library directories on fecode-bd for the current target. If the directories already exist nothing will be altered. This is useful when building for a previously undefined target.
- **make install** - Print help information about the make install rules.
- **make installscript** – Install all specified script files into the project's download directory.
- **make production** – Install the project's object file in the production library and its header files in the production reference files. Also create symbolic links from the production libraries to the test libraries. In the case of projects that are to be installed into download directories, the object module is placed in the download directory and a symbolic link is made from libxxx.out to testxxx.out. This rule effectively promotes the project from test to production.
- **make test** – Install the project's object file in the test library and its header files in the test reference files. In the case of projects that are to be installed into download directories, the object module is placed in the download as testxxx.out rather than libxxx.out.
- **make development** - Install the project's object file in the development library and its header files in the development reference files. In the case of projects that are to be installed into download directories, the object module is placed in the download as devxxx.out rather than libxxx.out.

A complete list of make rules can be produced by issuing the '**make help**' command.

## **6.0 Environment Variables**

The tools define and reference several environment variables. The variable name, location defined and a short description of each variable follows.

### **6.1 Developer Supplied Environment Variables**

The following environment variables are specified in the developer's `~/esd/scripts/useresdsetup.bash` file. The tools will provide standard default values if none are provided.

#### **USER\_ORGANIZATION**

Provides the name of the organization for which the tools are configured (e.g., rfies or pbares). This organization is used by the tools to properly locate include and library directories, and to search the repository for projects related to that organization.

#### **USER\_SCRIPT\_DIR**

Points to location where the tools can find (optional) developer supplied tool callback scripts.

#### **USER\_SANDBOX\_DIR**

Points to location where the tools can find sandbox directories for the developer's working set of projects. By convention each directory in the sandbox has the same name as the project that it contains.

#### **USER\_REPOSITORY**

Indicates which repository the developer wishes to use (e.g., CVS, SCCS or SCM.)

## **6.2 Utility Environment Variables**

The following environment variables are initialized by the tools for the convenience of the developer.

#### **USER\_PROJECT\_DIR**

Initialized in `setup.bash` – optionally tailored to developer requirements in developer's `project_projectName` callback script.

Points to the sandbox directory containing the developer's working copy of the current project.

## ESD\_BASE\_DIR

Initialized in esdsetup.bash.

Points to the base directory for tool operations. This directory contains subdirectories containing all tools, scripts and the SCM code repository.

## ESD\_DOWNLOAD\_DIR

Initialized in esdsetup.bash.

Points to the base directory of the project download directories. Each front-end project has a download directory containing the operating system, startup script and project object module to be downloaded at boot time.

## ESD\_SCRIPT\_DIR

Initialized in esdsetup.bash.

Points to the directory containing tool scripts.

## ESD\_DSP\_DIR

### ESD\_TESTDSP\_DIR

### ESD\_DEVDSP\_DIR

Initialized in esdsetup.bash.

Points to the base directory of the DSP loader libraries.

## ORG\_LIBDSP\_DIR

Initialized in esdsetup.bash.

Points to the base directory of the DSP shared libraries.

## ORG\_LIB\_DIR

### ORG\_TESTLIB\_DIR

### ORG\_DEVLIB\_DIR

Initialized in esdsetup.bash.

Points to the base directory of the project object libraries.

## ORG\_INC\_DIR

### ORG\_TESTINC\_DIR

### ORG\_DEVINC\_DIR

Initialized in esdsetup.bash.

Points to the directory containing the project public header files pool.

## PROJECT

Initialized in setup.bash – optionally tailored to developer requirements in developer's project\_projectName callback script.

Contains the name of the current project.

## TARGET

Initialized in setup.bash – optionally tailored to developer requirements in developer's project\_projectName callback script.

Contains the default target name for the current project.

A complete list of environment variable definitions can be obtained with the '**dumpesd**' command.

End.